CS 2051: Lambda Calculus

Anand Singh
Georgia Institute of Technology

Anson Tao
Georgia Institute of Technology

Syed Hamza Qadri
Georgia Institute of Technology

***Abstract*—In this paper, we explore the subject of lambda calculus, including its theory and applications. We formally define the basic syntax of lambda calculus and implement several common operators and constructs in the language. Additionally, we explore the area of Montague semantics, which is an application of lambda calculus to natural language processing. Finally, we present a basic lambda calculus interpreter in Python.**

I. Background

In the 1930s, as part of an exploration into the foundations of mathematics, American mathematician Alonzo Church introduced the lambda calculus, a formal system in logic for expressing computation through anonymous functions. Nearly a century later, in a world where computation is more important than ever, lambda calculus serves as a crucial foundation for the theory of programming languages and computer science in general. In this paper, we seek to explore its theory, rules, and applications.

At its core, lambda calculus is a simple and elegant set of rules for expressing functions as first-class values. What this means is that functions and variables both belong to the same class of values known as expressions and, thus, generally support the same operations. This will be explored in further detail below. Moreover, functions in lambda calculus are anonymous: that is, they are not given explicit names, which makes the semantics of the system simpler. By combining anonymous functions together, we are able to express any computable function; in other words, anything that can be computed can be represented through the rules of lambda calculus.

A. *Abstractions*

We have mentioned anonymous functions a few times up until now. In lambda calculus, these functions are known as abstractions. More specifically, an abstraction is an anonymous function that takes a single argument. Note that, although a single abstraction per se can only take a single argument, several abstractions can be combined through currying to effectively take multiple arguments.

> An abstraction is an expression of the form $\lambda x.M$, where x is a bound variable and M is another expression.

B. *Variables*

Variables are another kind of expression. In lambda calculus, a variable may be free or bound.

- **Bound variables** are essentially the parameters of abstractions, to be replaced by some other expression. They are restricted, or bound, to the scope of the abstraction within which they are defined. Bound variables are initially defined with a preceding λ character.
- **Free variables** are unbound and their value can vary. They may occur within abstractions but they are not restricted to the scope of the abstraction.

> - x is bound in the expression $\lambda x.M$.
> - If x is bound in the expression M or N then x is bound in the expression MN.

> - x is free in the expression x.
> - If x is free in the expression M and $x \neq y$, then x is free in the expression $\lambda y.M$.
> - If x is free in the expression M or N, then x is free in the expression MN.

C. *Application and β reduction*

Application is the process of applying an abstraction to an argument. It is done via β reduction, which refers to substituting a bound variable in an abstraction with some argument in the form of another expression, thereby simplifying the abstraction. Application is essentially the basis of computation in lambda calculus, as we will see later.

D. *Expressions*

With these basic concepts defined, we may proceed to define the expression, the overarching syntactical concept in lambda calculus. An expression, also known as a lambda term, represents a well-formed formula that can be computed. It is defined recursively in terms of variables, abstractions, and applications. The concept of the expression is central to the idea that functions are treated as first-class citizens in lambda calculus.

> - If x is a variable, then it is a valid expression itself.
> - If x is a variable and M is an expression, then $\lambda x.M$ is an expression.
> - If M and N are expressions, then MN is an expression.

E. *α equivalence*

α equivalence is the idea that the specific name of a bound variable is meaningless and can be changed. α equivalence allows us to rename bound variables to prevent clashes, such as during application or when currying.

F. η reduction

η reduction is the theoretical idea that two functions are the same if and only if they give the same results for all arguments, regardless of their implementation.

II. Main Result: The Power of Lambda Calculus

We have mentioned that any computable function can be represented through the language of lambda calculus. More formally, we may say that lambda calculus is Turing complete.

Turing completeness is a property of a system of rules that refers to the ability of that system to simulate a Turing machine. This is an abstract machine that solves problems by drawing and manipulating symbols on an infinite strip of tape. A key property of a Turing machine is that, despite the deliberate simplicity of the model, it is capable of implementing any computer algorithm; as such, a programming language that is Turing complete can theoretically solve any computable problem in some way (that is, given enough time and memory).

The formal proof for the Turing completeness of lambda calculus is not only involved (it is actually more common to prove the Turing completeness of other sets of rules by implementing a lambda calculus interpreter) but also not very helpful in explaining how lambda calculus can be used to solve more complex problems. In this section, we instead focus on defining common programming constructs to help the reader develop an intuition for how computable problems can be solved in lambda calculus. These constructs build upon the basic syntax that we have already defined.

A. Conditional branching

Conditional branching is analogous to the construct of 'if-statements' in programming languages. If a certain condition is satisfied, we evaluate a certain expression; otherwise, we evaluate another expression. We can represent this in lambda calculus as follows:

$$T = \lambda x.\lambda y.x$$
$$F = \lambda x.\lambda y.y$$
$$ifthenelse = \lambda i.\lambda t.\lambda e.ite$$

By defining booleans this way, we get conditional branching: if the argument i is T, then the expression t is evaluated; otherwise, if i is F, then the argument f is evaluated.

Consider the following example:

$(ifthenelse)Fxy$	Initial expression
$= (\lambda i.\lambda t.\lambda e.ite)Fxy$	Definition of $ifthenelse$
$= (\lambda t.\lambda e.Fte)xy$	β reduction
$= (\lambda e.Fxe)y$	β reduction
$= Fxy$	β reduction
$= (\lambda x.\lambda y.y)xy$	Definition of F
$= (\lambda a.\lambda b.b)xy$	α equivalence
$= (\lambda b.b)y$	β reduction
$= y$	β reduction

Since the boolean argument is F (false), the expression evaluates to the last argument (the "else" condition).

B. Logical operators

Note that logic operators can be represented in terms of booleans and if-statements. With this in mind, we can implement the basic logical operators in lambda calculus:

Operator	Pseudocode	Lambda expression
not x	if x then F else T	$(\lambda i.\lambda t.\lambda e.ite)xFT$
and x, y	if x then y else F	$(\lambda i.\lambda t.\lambda e.ite)xyF$
or x, y	if x then T else y	$(\lambda i.\lambda t.\lambda e.ite)xTy$

C. Arithmetic

Pure lambda calculus does not include arithmetic operators (or even numbers!) out of the box. Of course, since it is Turing complete, there must be some way to represent natural numbers. Indeed, the Church numerals are such a representation. They are defined as follows:

$$0 = \lambda f.\lambda x.x$$
$$1 = \lambda f.\lambda x.fx$$
$$2 = \lambda f.\lambda x.f(fx)$$
$$3 = \lambda f.\lambda x.f(f(fx))$$
$$...$$
$$n = \lambda f.\lambda x.f^n(x)$$

This notation makes it easy to find the successor of some number (i.e. 1 added to the number): simply wrap another f around the numeral. This gives us the successor function S:

$$S = \lambda n.\lambda f.\lambda x.f(nfx)$$

Since we can now add 1 to some number, we should be able to add any two natural numbers together through repeated application of the successor function. Indeed, we can define an addition function using the successor function:

$$A = \lambda m.\lambda n.nSm$$

Though we will not define every arithmetic operation (the definition for subtraction, in particular, is involved), the provided definitions should provide an intuition for how arithmetic works in lambda calculus.

D. Loops

In lambda calculus, a loop can be achieved using recursion. By extending the syntax to allow naming functions (as we have already done for explanation purposes above), it is easy to see how a recursive function could be implemented. Consider a function that takes a number n as an argument and adds all numbers between 0 and that number inclusive (assuming numerals, addition, and equality checking are already defined):

$$sum = \lambda n.((\lambda i.\lambda t.\lambda e.ite)(x == 0)(0)(n + sum(n - 1)))$$

The main challenge, however, is that the rules of (pure) lambda calculus do not allow for an abstraction to refer to itself inside its body. This is where fixed-point combinators come into play. In mathematics, a fixed point of a function is a point that maps back to itself. A crucial property of lambda

calculus is that every function has a fixed point, and we can define a function, known as a fixed-point combinator, that takes a function and maps it to its fixed point.

The Y combinator is perhaps the most well-known example of such a function. It is defined as follows:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Then, for all functions F, $YF = F(YF)$; that is, YF is a fixed point for any F.

This is all rather abstract, but it can be tied back to our initial problem: recursion. We can in fact rewrite any recursive function in terms of its fixed point, and then write it in terms of a fixed-point combinator. Take our sum function as an example. Using inverse β substitution (i.e. introducing a new bound variable), we can rewrite it as such:

$$sum = \lambda f.(\lambda n.((\lambda i.\lambda t.\lambda e.ite)(x == 0)(0)(n + f(n-1))))sum$$

Thus, sum is a fixed point of the function $\lambda f.(\lambda n.((\lambda i.\lambda t.\lambda e.ite)(x == 0)(0)(n + f(n-1))))$ and so we can rewrite the function in terms of the Y combinator:

$$sum = Y(\lambda f.(\lambda n.((\lambda i.\lambda t.\lambda e.ite)(x == 0)(0)(n + f(n-1)))))$$

Thus, we have defined a recursive function without referring to itself within its body, thereby adhering to the rules of lambda calculus. This proves that loops can be achieved in pure lambda calculus.

Recall that a language that can implement any algorithm (or, more formally, any computable function) is Turing complete. We have demonstrated how the basic building blocks of most algorithms can be represented in lambda calculus; as such, we have developed an intuition for the Turing-completeness of lambda calculus.

E. Application to functional programming

Now that we have implemented several programming constructs in lambda calculus, it is easy to see the relevance of lambda calculus to programming. One of the most important applications of lambda calculus to programming is through functional programming languages. Since lambda calculus is Turing complete, any computable function can be expressed in lambda calculus. Therefore, it serves well as a basis for programming languages, and the languages centered around lambda calculus are called functional programming languages.

The most important aspect of functional programming languages is the idea of pure functions. These functions create an output based solely on the input they are given - in other words, they do not keep track of state. Running a pure function with the same inputs one thousand times will still yield the same result as running the first time. Pure functions also have no side effects, meaning they do not have external impact beyond returning a value. For example, a setter method for a class attribute in Java would not be a pure function because it changes the state of the object it is called on.

Based on these rules, pure functions can be reduced to their input and output, and therefore its main complexity comes from how it is used in the system. However, it is difficult in practice to only have functions that do not affect a system, so typically programs will have functions that are not completely pure. Pure functions are much easier to implement in functional programming languages due to lambda calculus being the core of how the languages are processed.

Below we can an example of lambda calculus in Haskell, a functional programming language.

```
map (\x -> x + 1) [1, 2, 3]
```

The map function in Haskell takes in a function f and a list $[A]$ as two parameters, and applies f to every element in $[A]$ to get the output list $[B]$. In this case, map takes in a function $\backslash x \rightarrow x + 1$ that adds one to the input, the list $[1, 2, 3]$, and outputs the list $[2, 3, 4]$. Note how Haskell allows a function to be abstracted, allowing for reduction similar to lambda calculus. The equivalent in lambda calculus would be something like the following (assuming lists, digits, and addition are defined):

$$(\lambda x.(\lambda y.(map(x, y))))(\lambda z.(z + 1))([1, 2, 3])$$

Many non-functional programming languages also support functional programming. Below is an example of a function in Python that takes in a number x and function f and computes $x \cdot f(x)$.

```
multiply_function = lambda x, func: x * func(x)

# evaluates to 1000
multiply_function(10, lambda y: y * y)

# evaluates to 200
multiply_function(10, lambda z: z + z)
```

The equivalent of $multiply\_function$ in lambda calculus can be written as

$$\lambda x.x * f(x)$$

and the other two expressions can be written as follows:

$$\lambda x.(\lambda f.x * fx))(10)(\lambda y.y * y)$$

$$(\lambda x.(\lambda f.x * fx))(10)(\lambda z.z + z)$$

Due to the design of functional programming languages, there are many benefits of functional programming languages compared to other paradigms.

- Maintainability is much easier since functions don't have any external effects, so there is less coupling and systems are typically simpler and easier to understand.
- Debugging benefits from the same aspects. Since functions don't store state, share data outside of its scope, or

affect other functions or variables stored somewhere, it's easier to determine where a bug is coming from compared to Object-Oriented Programming languages.
- Modularity improves since functions can easily be composed without them affecting each other. The process of testing smaller functions and composing them to solve a larger, more complex problem is simpler to execute in functional programming languages.
- Concurrency is more natural due to separation of functions.

By understanding lambda calculus, we can better apply its properties and enhance our own programming, such as the ways lambda calculus provides many benefits to functional programming languages.

III. Generalization: Montague Semantics and Natural Language Processing

One exciting and interesting application of lambda calculus lies within natural language processing (NLP) for semantic analysis using the Montague semantics framework. Montague semantics aims to provide a formal treatment of real natural language meaning by using lambda calculus to represent the meaning of expressions.

Each sentence is assigned a lambda expression that represents its meaning. These lambda expressions can be combined using function application and abstraction to generate complex meanings for more complex sentences. The most basic example is the sentence "John loves Mary". In Montague semantics, this sentence can be represented as a lambda expression:

$$\lambda x.\lambda y.(love(x,y))(John, Mary)$$

This binds the variable x to John, and the variable y to Mary. The function love(x, y) takes two arguments and returns a truth value of "love" between the arguments.

A. *Montague Grammar*

Montague's original thesis was that natural languages and formal languages were two sides of the same mirror, and he fleshed this idea out through what became known as Montague grammar in a series of papers he wrote.

Montague grammar allows one to convert natural language phrases and sentences into applications of logical functions. Below is a table (not exhaustive) for converting natural language phrases into these functions, and a basic example.

Phrase type	Phrase	Meaning
Sentence	NP VP	(NP VP)
Verb Phrase	VP	$\lambda x.verb(x)$
Transitive Verb	TV	$\lambda y.\lambda x.transverb(x,y)$
Determiner	"Some" or "a"	$\lambda P.\lambda Q.\exists x(P(x) \wedge Q(x))$
Determiner	"Every"	$\lambda P.\lambda Q.\forall x(P(x) \rightarrow Q(x))$
Determiner	"No"	$\lambda P.\lambda Q.\nexists x(P(x) \rightarrow Q(x))$

B. *Examples*

Applying the determiner rule for "every" to the sentence "Every dog has bitten someone before" yields:

$$\forall x(Dog(x) \rightarrow hasBittenSomeone(x))$$

Then, we can further reduce $hasBittenSomeone$ using the verb phrase rule and determiner rule on "some" in "someone" to:

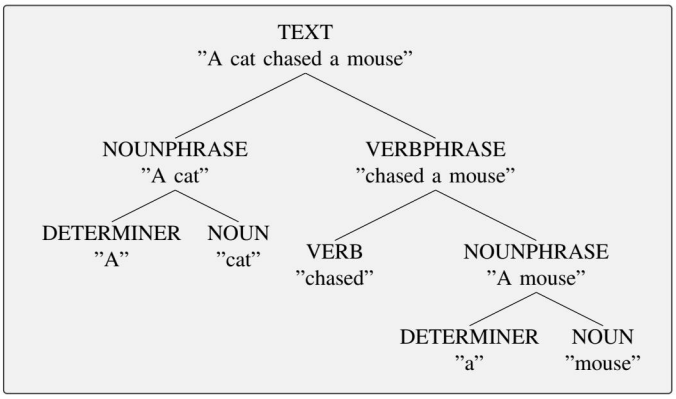
$$\forall x(Dog(x) \rightarrow \exists y(isPerson(y) \wedge bitten(x,y)))$$

Another example is applying the same rules to "A woman sees a man," using a $sees(x,y)$ function. This yields:

$$\forall x(isWoman(x) \rightarrow \exists y(isMan(y) \wedge sees(x,y)))$$

C. *Natural Language Processing Applications*

The above framework can be used to build semantic parsers that can extract meaning representations from text. These parsers use syntactic analysis to identify the grammatical structure of sentences and then use lambda calculus to generate the corresponding meaning representations, as per some examples above.



The above diagram shows one way a sentence could be reduced into a parse tree format, which then can be scanned and converted to lambda notation using our above table or rules. Here, we would apply the "a" rule for determiners twice for cat and mouse, and likely employ the use of a $chased(x,y)$ function. Note that there are likely many ways and rules of creating a parse tree and applying lambda calculus to the phrase types.

D. *NLTK in Python*

Natural Language Toolkit (NLTK) is a powerful Python library used for natural language processing, and includes functionality for working with logic as well as semantic parsing of sentences.

*1) **Propositional Logic**:* We can easily use the Proposition class to create various propositions using their operators. Below is an example of testing various inputs for P and Q to see if they satisfy P AND (Q OR NOT P).

```
from nltk.sem.logic import *

# Create the propositions
p1 = Proposition('P')
p2 = Proposition('Q')

# Create the expression
expression = And(p1, Or(p2, Neg(p1)))

# Create the valuation
v = Valuation([('P', True), ('Q', False)])

# Evaluate the expression under the valuation
result = expression.satisfy(v)
print(result)
```

*2) **First-Order Logic**:* We take another step towards evaluating natural languages by introducing first-order logic, which allows us to deal with quantifiers. These quantifiers will act on predicates which take some number of arguments, allowing us to represent phrases (For example, Aidan sees Andrew can be written as a predicate $sees(x, y)$ applied to $Aidan$ and $Andrew$ subjects).

*3) **Recursive Descent Parser**:* To evaluate the logical form of a sentence, we need to first parse the sentence and generate some representation of the sentence's structure. NLTK provides several parsers for generating parse trees, including the RecursiveDescentParser, the ShiftReduceParser, and the ChartParser.

IV. Lambda Calculus Interpreter: Code and Illustrations

To gain a deeper understanding of lambda calculus, we have developed a simple lambda calculus interpreter. By explaining our implementation, we hope to clearly explain and solidify the process by which an expression is evaluated.

We decided to adhere to some design choices and limitations:

- The interpreter is written in Python. Python's rich standard library allows for relatively simple string parsing, which makes it an ideal choice for this project. Also, Python's weak typing significantly simplifies the code for converting a string into a tree, and may also make it easier to extend the syntax later.
- The interpreter understands pure lambda calculus only. This means that there is no support for things like arithmetic, equality checking, or even numbers out of the box (though, of course, they can be implemented in the lambda calculus language, as explained earlier). Implementing these features from the beginning would detract from the actual purpose of our project, which is to better understand lambda calculus.
- The only modification we have made to the standard lambda calculus syntax is to require the use of a backslash in place of a lambda character, as this is easier to type using a standard keyboard.

At a high level, the project is made up of three distinct components; these components work after one another to deconstruct the expression from a string into a machine-readable format, evaluate it in this format as much as possible, and then reconstruct it into a string.

We will use the following example to illustrate the functionality of these components:

$$(\lambda a.\lambda b.a)(\lambda c.d)(\lambda e.f)$$

A. *Lexer*

Our initial lambda expression input is a string. Strings are cumbersome to work with as they are not easily machine-readable. Thus, our first step is to convert the string into a series of tokens. This task is done by the lexer (lexical analyzer) component.

Our tokens represent the most basic syntactical elements of an expression, namely:

- Abstraction headers (the "$\lambda x.$" section)
- Variables
- Parentheses

Here is our lexer's main function:

```
def lex(ss: StringStream) -> TokenStream:
    tokens = []
    while not ss.eof():
        c = ss.next()
        if c == ' ':
            continue
        elif c == '\\':
            tokens.append(Token(TokenType.
                ABSTRACTION_HEADER, ss.next()))
            if ss.next() != '.':
                raise ValueError("Syntax error
                    ")
        elif (c >= 'A' and c <= 'Z') or (c >= '
            a' and c <= 'z'):
            tokens.append(Token(TokenType.
                VARIABLE, c))
        elif c in {'(', ')'}:
            tokens.append(Token(TokenType.
                PARENTHESIS, c))
        else:
            raise ValueError("Invalid character
                : " + c)
    return TokenStream(tokens)
```

Consider our example. The lexer would break it down into the following components:

```
PARENTHESIS: (
ABSTRACTION_HEADER: a
ABSTRACTION_HEADER: b
VARIABLE: a
PARENTHESIS: )
PARENTHESIS: (
ABSTRACTION HEADER: c
VARIABLE: d
PARENTHESIS: )
PARENTHESIS: (
ABSTRACTION_HEADER: e
VARIABLE: f
PARENTHESIS: )
```

B. *Parser*

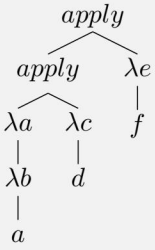
With lexing complete, we move on to perhaps the most crucial component of the interpreter: the parser. The parser analyzes the tokens, determines relationships between them, and ensures that they are well-formed. In simple terms, while the lexer operates at the word level, the parser operates at the grammatical level.

Specifically, the parser works by transforming the tokens into an abstract syntax tree (AST). This is a data structure commonly used by compilers and interpreters that represents source code in the form of a tree, where each node represents some programming construct occurring in the code. An AST is a far more "machine-readable" format than a string, as it can be evaluated recursively.

Fortunately, the AST of a lambda expression is remarkably simple: it is a binary tree that is defined simply in terms of variables, abstractions, and applications.

- An application node denotes that the abstraction at its left child is to be applied to the argument at its right child.
- An abstraction begins with a lambda node - for example, λx. The body of the abstraction is defined by the node's left (and only) child.
- A variable is simply defined by a single node. The leaves (nodes with no children) of the AST are always variables.

Here is the abstract syntax tree representation of our earlier example:



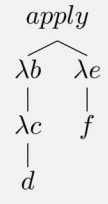
The code for generating these abstract syntax trees is relatively involved and the most complex section of the project. In our implementation, we begin by passing through the tokens once to separate sub-expressions into separate sublists; then, we recursively separate these sublists into applications, making sure to handle abstractions appropriately. While this may not be the most efficient method as it requires two passes, it is easy to implement, and the code is quite readable as well. Note that Python's weak typing is especically useful here: our parser function is able to accept either a list of tokens or a token itself (as the base case), and lists themselves may contain tokens or further sublists.

```
def first_pass(tokens: TokenStream) -> list[
    Token]:
    buffers = [[]]
    while not tokens.eof():
        token = tokens.next()
        if token.type == TokenType.PARENTHESIS:
            if token.value == '(':
                buffers.append([])
            else:
                buffers[-2].append(buffers[-1])
                buffers.pop(-1)
        else:
            buffers[-1].append(token)
    return buffers[0]

def parse(expr: Token | list[Token]) -> ASTNode
    :
    if isinstance(expr, Token):
        assert expr.type == TokenType.VARIABLE
        return ASTNode(ASTNodeType.VARIABLE,
            expr.value)

    if len(expr) == 1:
        return parse(expr[0])

    if isinstance(expr[0], Token) and expr[0].
        type == TokenType.ABSTRACTION_HEADER:
        node = ASTNode(ASTNodeType.ABSTRACTION,
             expr[0].value)
        node.left = parse(expr[1:])
        return node

    node = ASTNode(ASTNodeType.APPLICATION)
    node.left = parse(expr[:-1])
    node.right = parse(expr[-1])
    return node
```

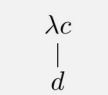
C. *Interpreter*

Finally, we have the interpreter itself. The interpreter traverses through the AST, performing every possible application until no more applications are possible. Then, the AST is directly reconstructed into a string, which is the evaluated form of the given expression.

Consider our example again. The interpreter would begin by performing the application at the left child of the root node, deleting the λa node and replacing the a node with the λc node. This would result in the following AST:



Next, the interpreter would complete the application at the root node in the same manner, resulting in the following:



All possible applications would be completed at this point, so the AST would be reassembled into a string:

$$\lambda c.d$$

Indeed, this is the evaluated form of our initial expression.

The code for this section is recursive in nature as well. We begin by recursing into the left and right children of the current node; then, if the current node is an application that can be evaluated, we do so by replacing all instances of the bound variable with the argument. Finally, we recurse into the children again to perform any applications that were not possible earlier.

```
if root is None:
        return root

    root.left = interpret(root.left)
    root.right = interpret(root.right)

    if root.type == ASTNodeType.APPLICATION and
         root.left.type == ASTNodeType.
        ABSTRACTION:
        bound_var = root.left.value
        replacement = root.right
        root = find_and_replace(
            root.left.left,
            lambda node: node.type ==
                ASTNodeType.VARIABLE and node.
                value == bound_var,
            lambda node: replacement
        )
        root.left = interpret(root.left)
        root.right = interpret(root.right)

    return root
```

V. Looking Ahead: Challenges and Reflection

A. *Future Extensions*

One topic we wanted to explore with more time is combinatorial logic. In combinatorial logic, lambda expressions are simplified by substituting various parts with combinators (functions without free variables). This makes reduction much simpler and it is easy to transform lambda calculus into combinator expressions. It is used to model hardware and some functional programming languages, and is useful for theoretical models of computation.

B. *Interpreter*

Our interpreter still has considerable room for improvement. Within the scope of pure lambda calculus, we still need to implement α equivalence by renaming variables in the event of a clash. Additionally, we should also allow the user to set reusable variables (perhaps denoted by capital letters) to make code reuse and functional decomposition easier. Outside pure syntax, we could also implement digits, arithmetic, and logic out of the box for the user's convenience, as an optional feature. We could also perhaps add a GUI component to visualize the evaluation process, either in terms of the expression itself or its AST. Better error handling is needed as well, as we currently only minimally check for errors at the lexer stage.

C. *Reflection*

Considering all our findings, it is clear that the strength of lambda calculus lies in its ability to represent any construct, algorithm, or data structure in terms of relationships between simple single-argument functions, in a way that is more expressive and human-readable than other formalisms like the Turing machine. This is what makes it ideal as a basis for programming languages, as well as for other applications such as natural language processing where relationships between terms are important. It also has useful properties such as acting on inputs in consistent ways and only having effects on systems through the return values, which makes programming languages centered around lambda calculus (functional programming languages) much simpler and easier to work with in certain aspects.

References

- Brilliant Math and Science Wiki: Lambda Calculus - This website rigorously yet clearly defines the syntax of lambda calculus as well as how to express common constructs in it. It was a very useful reference for our main result.
- Stanford Library of Philosophy: The Lambda Calculus - This is another introduction to lambda calculus that explains some aspects of its usage in more detail and also introduced us to our generalization topic.
- Jason Eisner: Functional Programming: This document explains the relationship between functional programming and lambda calculus. It allowed us to understand how lambda calculus serves as the basis for the functional programming paradigm, and programming languages in general.
- Stanford Library of Philosophy: Montague Semantics - This website, also from the Stanford Library of Philosophy, offered a very detailed introduction to and description of Montague semantics that we referred to extensively.
- Richard Montague: The Proper Treatment of Quantification in Ordinary English - This is a relevant research paper from Richard Montague, the mathematician who pubished what became known as Montague semantics. As such, it was very useful for us.
- NLTK: Sample usage for logic - This is the official documentation for Python's NLTK library; as such, it was a useful reference when writing the section on NLTK.
- Lisperator: Writing a parser - This website served as a good (albeit generic) guide for how to develop interpreters, and gave us sufficient direction to begin writing our lexer and parser.
- University of Wisconsin: Lambda Calculus - This document introduced us to abstract syntax trees for lambda calculus. Though our ASTs are slightly different from the examples given here, this page still served as a good introduction.